The purpose of this worksheet is, to provide the definition of a very basic, 2x2 rotation matrix, based on a presumably known angle theta. According to Linear Algebra, it's possible to define two 2-element vectors, like so:

$$X = \begin{pmatrix} x \\ y \end{pmatrix}, \; X' = \begin{pmatrix} x' \\ y' \end{pmatrix}$$

Thereby, again according to Linear Algebra, if a column-vector is multiplied by a matrix, to result in another column-vector, this can be written like so:

$$X' = \begin{pmatrix} a & b \\ c & d \end{pmatrix} X$$

But then, if [a .. d] are known constants, the way to compue X' follows like so:

$$x' = a\,x + b\,y$$
$$y' = c\,x + d\,y$$

But, according to Trigonometry, if a point is defined as having coordinates (X), from which the coordinates (X') are to result, when rotated through angle theta, then this would be the way to compute the resulting coordinates individually:

$$x' = \cos\theta\,x - \sin\theta\,y$$
$$y' = \sin\theta\,x + \cos\theta\,y$$

But then, causal inspection of these equations reveals, that both x' and y' are individually linear combinations of x and y. And so, the actual matrix equation which follows is:

$$X' = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} X$$

Therefore, if the angle of rotation is known, then by filling the matrix with elements, which are the correctly-chosen and correctly-signed trig functions of that rotation angle, a matrix can be computed which will rotate any 2-dimensional coordinate vector. Further, this concept can be extended to 3-dimensional coordinate vectors, as long as the method used to define the rotation yields corresponding elements of a 3x3 matrix. And again the result will follow, because each out of 3 resulting coordinates, will individually be a linear combination of the 3 original coordinates.

The application to 3D (solid) angles becomes more tricky, because more than one system exists to state the angle, as composed of 3 values. One such system is the "Euler Angle", and another is a sequence of virtual rotations, once around (x), again around (y), and finally around (z). Either way, the order of the coordinates becomes important. But the second of these two systems becomes plausible sooner, because 3 matrices can be written, each stating a different 2D rotation, and the matrix product is then easy to conceive.

A side-effect with 3D rotation matrices is, that at least some of the elements must form as products and sums, of the trig functions, of more than one angle-component.

If the goal of the exercise is to translate a Euler Angle into a 3x3 rotation matrix, then the task could be given as follows:

- The Euler Angle *could* regard the Y-axis as 'vertical', and *could* regard the X-axis as the original axis which, after being 'Panned' around the Y-axis, is to be 'Tilted' Up or Down, after which the coordinates are to be 'Rolled'.
- This can be re-represented as 3 separate rotations around the Fixed-Axis-Sequence 'XZY'. In other words, the order of the axes of Pan, Tilt and Roll can simply be reversed.
- Each rotation around a fixed axis can be translated into a 3x3 matrix, that leaves 1 axis unchanged, after which the matrix product can be computed.
- Such arithmetic simplifications are also a reason, for which certain 3D graphics software simply offers the user a single type of solid angle, but with the axes in different orders.
- In order to keep the usage comprehensible, software which does this will also abide by a convention which defines the direction of each rotation, such as 'Counter-Clockwise, Facing the Origin'.

The matrices which result would be as follows, assuming a Right-Handed Coordinate System:

$$\text{RX: } \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_X & -\sin\theta_X \\ 0 & \sin\theta_X & \cos\theta_X \end{pmatrix}$$

$$\text{RY: } \begin{pmatrix} \cos\theta_Y & 0 & \sin\theta_Y \\ 0 & 1 & 0 \\ -\sin\theta_Y & 0 & \cos\theta_Y \end{pmatrix}$$

$$\text{RZ: } \begin{pmatrix} \cos\theta_Z & -\sin\theta_Z & 0 \\ \sin\theta_Z & \cos\theta_Z & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

(Assuming 'XZY'...)

$$M = RY\ RZ\ RX$$

There exists a family of rotation-angle definitions, which cannot be converted into matrix representation as easily as the above, and those would be rotation angles, which have as their preferred axes, arbitrary direction vectors. At the same time, Quaternions would be more-difficult to convert into matrices, based on straightforward analysis such as what is written above. As an example of the first family of rotation-angles, a convention also exists which states a vector as the single, arbitrary axis of rotation, and which either uses the magnitude of the vector, or a separately-supplied real number, to state by what angle the rotation is to take place, as if looking back at the origin from the stated direction.

The reader may find the formulae to convert those types of rotation-definitions into 3x3 matrices, on his or her own time.

One place in which this issue must crucially be solved however, is in the 'Rigging' of 3D models for CGI, where each model possesses numerous joints, and where the movement of any joint might best be made familiar to the content-designer, as a Euler Angle, that has arbitrary preferred axes, which at first glance, might just be arbitrary direction vectors in the model's own coordinates.

But, in the case where 3D models are rigged, and are given an internal skeleton, there is a saving priniciple which generally applies. The internal skeleton of the 3D model is hierarchical in nature. This means that each (simulated) bone connects to a parent bone, so that according to "Forward Kinematics", rotating the parent bone, also rotates all the attatched child-bones. Forward Kinematics needs to be understood, before 'Inverse Kinematics' can be understood.

But in that context, it can be required by the 3D rendering system, that the 'root' bone - the parent of all the model's bones - must always have a fixed orientation with respect to the model's coordinates. The intial rotation of bones, may start out as relative to the model's coordinates. And this can also result in the child bones receiving rotated axes for their Euler Angles. But this idea of Forward Kinematics also requires that Math be applied in correct sequence, to convert the rotation matrices that are local to any one child bone, into rotation matrices which state that child bone's rotation in a way global to the model. It follows that the global matrix for any child bone, forms as the matrix product of the entire chain of parent bones, all the way back to the root bone.

In order for that approach really to work however, the concept must also be introduced, that each bone have 'a *chained*, neutral angle', in which they don't displace their vertices, but for which the *inverse* matrices have been computed.[1] Each bone's *chained*, animated angle would need to be multiplied by its inverse, neutral matrix, to arrive at the current bone's global matrix.

Such approaches are again, not meant to complicate the design of CGI software, which must at least feed global rotation matrices to the GPU, or which must feed animated vertex-positions to the GPU. Rather, such approaches actually simplify the implementation of certain 3D rendering engines, because they reduce the methodologies to one predictable methodlogy, that will work, regardless of how deeply convoluted the content becomes, which the content designer creates. If the 3D rendering engine is to supply alternatives, each time, the alternatives also need to be fleshed out, until a content designer can apply them to whatever level of complexity he or she wishes. Only then, has a 3D rendering engine been coded correctly.

If the application is CGI-related, then there is an added level of complexity (which might act as a deterrent against the coders' adding their own complexity). Here, transformation is defined as a rotation, followed by a translation. The use of stretching and scaling, by the rotation matrix, has often been abandoned in the design of 3D rendering engines, because to integrate scaling with the rotation matrix, already makes the project too cumbersome to implement.

The displacement vector often needs to be computed in an inverted way.

If the rotation matrices are to be applied to the skinning of a 3D model, then the basic assumption is that the rotation by itself will affect the position of eventual vertices, relative to the origin of the model (which was also how they were stored). But in the case of bones, what's really desired is that the position of the hinge-point, at which each bone connects to the parent bone, not move, due to the rotation of the current bone, that the hinge-point anchors.

The systematic way to achieve that would be, first to compute by how much the rotation of any one bone, also displaces its hinge-point, the position of which needs to be stored in a memory location used by the CPU. And this internal representation of the position of the hinge-point, is in model coordinates. *The negative of this displacement vector* needs to be stored as the displacement of the single bone, that accompanies its rotation.

[1] In such cases, instead of actually computing a matrix-inverse, one inverts the angle.

Similarly, the goal could be for the CPU to keep recomputing the View Matrix, which transforms World Coordinates, into Camera Coordinates, aka View Coordinates. This matrix needs to rotate world coordinates in the inverted angles, from the angles in which the camera is panned, tilted and rolled, and *the inverse of the camera's origin vector* must then be put through the resulting rotation, to result in the displacement. This is because the world center is effectively being oriented.[2]

But along with the View Matrix, any real 3D rendering engine also needs to compute the Inverse View Matrix, for various purposes that go beyond the scope of this document to explain. This Inverse View Matrix converts from View Coordinates, back into World Coordinates. The most practical way to compute this matrix is *not*, to compute the View Matrix first, and then to invert that.

Instead, the most efficient way to compute the Inverse View Matrix is, to use the camera's pan, tilt and view angles (Euler Angles) directly for the associated rotation matrix. Then, the camera's position vector can be used directly as the displacement, because it's being added to whatever view coordinates are given, to arrive at world coordinates.

Dirk Mittler

[2]Such inversion of Euler Angles also requires the reversal of the order in which axes are applied.